Self-decomposable Global Constraints

Jean-Guillaume Fages and **Xavier Lorca** and **Thierry Petit** [1]

Abstract. Scalability becomes more and more critical to decision support technologies. In order to address this issue in Constraint Programming, we introduce the family of self-decomposable constraints. These constraints can be satisfied by applying their own filtering algorithms on variable subsets only. We introduce a generic framework which dynamically decompose propagation, by filtering over variable subsets. Our experiments over the CUMULATIVE constraint illustrate the practical relevance of self-decomposition.

1 INTRODUCTION

The complex nature of industrial problems brings opportunities to decision support technologies. One of them is Constraint Programming (CP). This technology has been successfully used in industry since the early 90's. However, the data explosion has raised the scalability issue of CP.

The resolution of a CP model can roughly be represented as a loop involving filtering algorithms to provide inference and a search process to perform hypothesis. This loop is iterated until a solution is found or, in case of an optimization problem, a solution is found and proved to be optimal. Filtering can be ensured by either global constraints or generic decompositions [4, 6]. The first option is usually advised to better solve problems. However, for a large value of n, even an $O(n \log n)$ time complexity can be too much, because constraints may be propagated several times at each node of a search process [15]. In this situation, it seems necessary to somehow reduce the value of n. As the time complexity of a propagator often depends on the number of variables it involves, this means filtering over less variables. The idea of filtering over subsets of variables is not brand new. For instance, it has been used in [2] in order to reduce the runtime of the SOMEDIFFERENT propagator, which is exponential in the worst case. However, to the best of our knowledge, such approaches have always been performed in an *ad hoc* way. The purpose of this article is to investigate a generic class of constraints on which subset filtering applies. We formalize two well known decomposition patterns ($f^{\cap}_{cc}$ and f^{val}) and extend them to get new tradeoff possibilities between filtering and runtime ($f^{\cap}_{vn}$ and $f^{val}_{\mathcal{P}}$). Our paradigm is independent from the algorithm used, and it requires no modification of the latter. As a consequence, it can be used in order to augment the scalability of existing propagators in CP tools.

First, we define in Section 2 self-decomposable constraints, a large family of constraints which hold on some of their variable subsets. Second, we introduce in Section 3 a generic framework, which uses the structure of variable domains to allow a light propagation of self-decomposable constraints. Third, Section 4 discusses and experimentally evaluates this feature in the context of cumulative scheduling. Finally, a conclusion and openings are given in Section 5.

[1] TASC - Ecole des Mines de Nantes, email: {firstname.name}@mines-nantes.fr

2 THEORETICAL FUNDAMENTALS

A Constraint Network [9] is defined by a set of variables and a set of constraints [4]. Each variable has a finite domain of integer values. Each constraint c defines a set of allowed combinations of values for the variables in its scope. We use the notation $c(\mathcal{X})$ to denote an instance of the constraint c which is defined over the variable set $\mathcal{X}$. A combination of values τ for the variables in $\mathcal{X}$ is valid if and only if each value in τ belongs to the domain of its variable. We use the notation $SAT(c(\tau))$ to state that τ satisfies c, that is, τ is an allowed combination of values for c. We use the notation $SAT(c(\mathcal{X}))$ to state that, given the domains of the variables in $\mathcal{X}$, there exists a valid combination τ such that $SAT(c(\tau))$. For sake of clarity, we assume c is equipped with a unique filtering algorithm. A filtering algorithm is a procedure that removes from domains values that do not belong to any solution of the constraint. Moreover, in this paper, when referring to variables we implicitly consider their respective domains as well. We use the notation $\mathcal{P}(\mathcal{X})$ to represent all subsets of $\mathcal{X}$.

Prior to debate the practical interest which stems from filtering variable subsets only, it is important to be sure such a filtering cannot violate the constraint. For this, we reformulate in Definition 1 the definition of *constraint monotonicity* given by Barták [3] and exploited by Maher [8]. This concept was introduced to study dynamic global constraints, whose set of variables can grow during search. As we study the case where propagation is performed on variable subsets, our equivalent formulation takes an opposite perspective on constraint monotonicity.

Definition 1 *A constraint $c(\mathcal{X})$ is **monotonic** if and only if for any variable subset $\mathcal{Y} \in \mathcal{P}(\mathcal{X})$, if $c(\mathcal{X})$ is satisfiable then its projection on $\mathcal{Y}$ remains satisfiable, that is, $SAT(c(\mathcal{X})) \Rightarrow SAT(c(\mathcal{Y}))$.*

Note that different monotonicity definitions can be found in the literature, such as the constraint monotonicity of [14], the propagator monotonicity [12] and the Boolean circuit monotonicity [6].

2.1 Constraint subset-monotonicity

As monotonic constraints hold on every subset of their variables (Definition 1), they can be propagated over any variable subset without loosing any solution (Proposition 1).

Proposition 1 *If $c(\mathcal{X})$ is monotonic, then applying its filtering algorithm to any variable subset $\mathcal{Y} \in \mathcal{P}(\mathcal{X})$ leads to no solution lost.*

Proof: Assume that a solution of $c(\mathcal{X})$ is lost by applying the filtering algorithm of $c(\mathcal{Y})$. Let s be such a solution. Consider now that we reduce the domain of each variable x_i in $\mathcal{X}$ to the value taken by x_i in s. Using such new domains, by construction $c(\mathcal{X})$ is satisfied while $c(\mathcal{Y})$ is violated, a contradiction with Definition 1. □

However, as pointed out in [3], the monotonicity is a strong requirement which only a few constraints fit. We then introduce a multifunction f to enumerate *some* variable subsets, *i.e.,* $f(\mathcal{X}) \subseteq \mathcal{P}(\mathcal{X})$, to generalize monotonicity (Definition 2).

Definition 2 *A constraint $c(\mathcal{X})$ is **f-monotonic** if and only if it holds on every subset of $\mathcal{X}$ induced by f, i.e., for any $\mathcal{Y} \in f(\mathcal{X})$, $SAT(c(\mathcal{X})) \Rightarrow SAT(c(\mathcal{Y}))$.*

Proposition 2 *If $c(\mathcal{X})$ is f-monotonic, then applying its filtering algorithm to any variable subset $\mathcal{Y} \in f(\mathcal{X})$ leads to no solution lost.*

We omit the proof which is similar to the case of Proposition 1. If we note $f_\forall$ the multifunction which enumerates all subsets, monotonicity is equivalent to $f_\forall$-monotonicity. Moreover, let f_I be the identity multifunction, *i.e.,* for any $\mathcal{X}$, $f(\mathcal{X}) = \{\mathcal{X}\}$, then any constraint is f_I-monotonic.

2.2 Constraint subset-decomposition

We have characterized constraints which can be propagated over variable subsets without loosing any solution. Now, we study the case where doing so is sufficient to satisfy the constraint, *i.e.,* filtering this constraint over every subset in $f(\mathcal{X})$ only does not produce any false solution. For that purpose, we introduce the definition of f-decomposition (Definition 3).

Definition 3 *A constraint $c(\mathcal{X})$ is **f-decomposable** if and only if the solution set of $c(\mathcal{X})$ is equal to the solution set of $\bigwedge_{\mathcal{Y} \in f(\mathcal{X})} c(\mathcal{Y})$.*

It is worth noticing that, if a constraint c is f-decomposable, then c is f-monotonic as well, but the opposite is not always true.

Up to now, we can use f-decomposition in order to satisfy constraints fast, but this decomposition may result in a lack of filtering. We now introduce the definition of *consistency preservation* (Definition 4) which means that a given level of consistency can be ensured while filtering over some variable subsets only.

Definition 4 *Given any consistency level λ, an f-decomposable constraint $c(\mathcal{X})$ is **f-λ-preserving** if and only if λ-consistency over every subset of $f(\mathcal{X})$ establishes λ-consistency over $\mathcal{X}$.*

Self-decomposable constraints (Definition 5) form the general class of constraints for which subset filtering applies, without any modification of that algorithm.

Definition 5 *A constraint is **self-decomposable** if and only if there exists a subset enumeration multifunction $f \neq f_I$, for which the constraint is f-decomposable.*

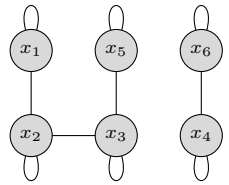
We use the concept of *intersection graph* to generate some remarkable subset enumeration multifunctions. We recall that the intersection graph of a set of variables represent the possible intersections of their respective domains (Definition 6). An illustration is provided in Figure 1. We now formally introduce several general self-decompositions, which will be illustrated over ALLDIFFERENT($\mathcal{X}$).

Definition 6 *Given a set of variables $\mathcal{X}$, the intersection graph $G = (V, E)$ of $\mathcal{X}$ represents the relation of domain intersections on $\mathcal{X}$. Each variable $x \in \mathcal{X}$ is associated with a vertex $v \in V$ and there is an edge $(i, j) \in E$ if and only if $dom(x_i) \cap dom(x_j) \neq \emptyset$ (with possibly $i = j$).*

$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$

$domain(x_1) = \{0\}$
$domain(x_2) = \{0, 1\}$
$domain(x_3) = \{1, 3\}$
$domain(x_4) = \{4, 5\}$
$domain(x_5) = \{2, 3\}$
$domain(x_6) = \{5\}$

(a) Variables and domains

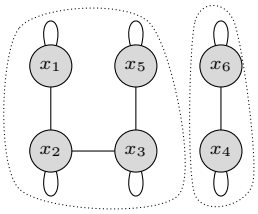


(b) Intersection graph

Figure 1: Intersection graph illustration.

2.2.1 Connected components-based decomposition

Let $f_{cc}^{\cap}$ be the multifunction which enumerates subsets associated with every maximal connected component in the intersection graph of the variable set in parameter. It is well known that ALLDIFFERENT is $f_{cc}^{\cap}$-decomposable, because two connected components form independent matching problems. Furthermore, a kind of $f_{cc}^{\cap}$-decomposition is described in [2] as the most successful trick in the implementation of SOMEDIFFERENT. On our leading example, $f_{cc}^{\cap}(\mathcal{X}) = \{\{x_1, x_2, x_3, x_5\}, \{x_4, x_6\}\}$ (Figure 2). Therefore, the $f_{cc}^{\cap}$-decomposition of ALLDIFFERENT($\mathcal{X}$) implies to filter ALLDIFFERENT($\{x_1, x_2, x_3, x_5\}$) and ALLDIFFERENT($\{x_4, x_6\}$) separately. Note that $f_{cc}^{\cap}$-decomposition leads to at most $|\mathcal{X}|$ subsets.



$f_{cc}^{\cap}(\mathcal{X}) = \{\{x_1, x_2, x_3, x_5\}, \{x_4, x_6\}\}$

Figure 2: $f_{cc}^{\cap}$ illustration.

An interesting property of $f_{cc}^{\cap}$-decomposition is that it can preserve the filtering quality. Theorem 1 provides the formal proof for the preservation of Generalized Arc Consistency (GAC). This can be directly adapted to Bound Consistency.

Theorem 1 *Any $f_{cc}^{\cap}$-decomposable constraint is $f_{cc}^{\cap}$-GAC-preserving.*

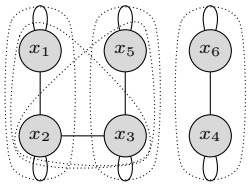
Proof: Assume a constraint c is $f_{cc}^{\cap}$-decomposable and not $f_{cc}^{\cap}$-GAC-preserving. "Not $f_{cc}^{\cap}$-GAC-preserving" means that there exists a value v in the domain of a variable x in a set $\mathcal{Y}_i$ that is not removed by $c(\mathcal{Y}_i)$ while it is removed by $c(\mathcal{X})$. As the consistency is GAC, there exists a solution of $c(\mathcal{Y}_i)$ with $x = v$. Therefore, to ensure that c is $f_{cc}^{\cap}$-decomposable, another set $\mathcal{Y}_j \neq \mathcal{Y}_i$ should contain x, so that $c(\mathcal{Y}_j)$ removes v from the domain of x. This hypothesis is absurd because by construction of the connected components in the intersection graph, $f_{cc}^{\cap}(\mathcal{X})$ forms a partition of $\mathcal{X}$. □

Depending on the complexity of the filtering algorithm, this decomposition may result in runtime improvement. However, from an operational point of view, $f_{cc}^{\cap}$-decomposition only makes a difference when the intersection graph of variables has many, balanced in size, maximal connected components. Unfortunately, this is quite a strong assumption in the general case. Therefore, we then introduce the vertex neighborhood-based decomposition, which provides presumably smaller subsets.

2.2.2 Vertex neighborhood-based decomposition

Let $f^{\cap}_{vn}$ be the multifunction which enumerates variable subsets associated with every vertex neighborhood in the intersection graph of the variable set in parameter (Figure 3). On our example, the $f^{\cap}_{vn}$-decomposition implies to filter ALLDIFFERENT over subsets $\{x_1, x_2\}, \{x_1, x_2, x_3\}, ..., \{x_4, x_6\}$ separately. We thus have more subsets than in the $f^{\cap}_{cc}$-decomposition, but they are smaller. In general, the number of variable subsets stemming from an $f^{\cap}_{vn}$-decomposition is exactly $|\mathcal{X}|$ and they may be duplication.

$f^{\cap}_{vn}$-decomposition allows a lighter propagation of $f^{\cap}_{cc}$-decomposable constraints (Theorem 2), while preserving their solution set. Presumably, the sparser the intersection graph, the faster propagation. Furthermore, it enables to make a filtering algorithm incremental, by skipping neighborhoods associated with variables for which no domain modification has occurred since the last propagation. However, this potential speedup comes at a price: the filtering of the $f^{\cap}_{vn}$-decomposition may be weaker than the filtering of the $f^{\cap}_{cc}$-decomposition (Theorem 3).



$f^{\cap}_{vn}(\mathcal{X}) = \{\{x_1, x_2\}, \{x_1, x_2, x_3\}, \{x_2, x_3, x_5\}, \{x_4, x_6\}, \{x_3, x_5\}, \{x_4, x_6\}\}$

Figure 3: $f^{\cap}_{vn}$ illustration.

Theorem 2 *$f^{\cap}_{cc}$-decomposable constraints are $f^{\cap}_{vn}$-decomposable, and conversely.*

Proof: In any fully instantiated solution, any connected component of the intersection graph is a clique. This means that each vertex neighborhood is a maximal connected component, so $f^{\cap}_{cc}$ and $f^{\cap}_{vn}$ are equivalent. □

Theorem 3 *$f^{\cap}_{vn}$-decomposable constraints are not all $f^{\cap}_{vn}$-GAC-preserving.*

Proof: Let us consider a variable set $\mathcal{X} = \{x_1, ..., x_7\}$, with respective domains $\{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{5, 6\}, \{5, 6\}, \{2, 5\}\}$, and a set of disequalities $\mathcal{C} = \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_4, x_4 \neq x_5, x_5 \neq x_6, x_6 \neq x_7, x_3 \neq x_6, x_7 \neq x_1\}$. We can use SOMEDIFFERENT$(\mathcal{X}, \mathcal{C})$ to have a global point of view over $\mathcal{C}$. The GAC of SOMEDIFFERENT$(\mathcal{X}, \mathcal{C})$ removes the value 2 from the domain of x_1, whereas its $f^{\cap}_{vn}$-decomposition does not. Therefore, $f^{\cap}_{vn}$-decomposition does not preserve GAC in the general case. □

2.2.3 Value-based decompositions

Let $\mathcal{D}$ and $\mathcal{X}_v$ respectively denote the union of variable domains and the variable subset induced by a value v in $\mathcal{D}$. In other words, $\mathcal{D} = \bigcup_{x \in \mathcal{X}} domain(x)$ and $\mathcal{X}_v = \{x \in \mathcal{X} \mid v \in domain(x)\}$, for any value v in $\mathcal{D}$. We introduce the multifunction f^{val} to provide the variable subset associated with every value. More precisely, $f^{val}(\mathcal{X}) = \{\mathcal{X}_v \mid v \in \mathcal{D}\}$, so $|f^{val}(\mathcal{X})| = |\mathcal{D}|$. On our leading example (Figure 4a), the f^{val}-decomposition of implies to filter ALLDIFFERENT on subsets $\{x_1, x_2\}, \{x_2, x_3\}, ..., \{x_4, x_6\}$.

Such a decomposition can be used for several other constraints, such as GCC [11], which restricts the number of occurrences of every value, and BINPACKING, its weighted generalization: Whenever only maximum occurrences and maximum bin capacities are considered, then these constraints are f^{val}-decomposable. It is worth noticing that their f^{val}-decomposition amount to their usual static decomposition, which are respectively based on one OCCURRENCE and one SCALAR constraint for each value and bin. From Theorem 4, these constraints are $f^{\cap}_{vn}$-decomposable as well.

Theorem 4 *f^{val}-decomposable constraints are $f^{\cap}_{vn}$-decomposable, and conversely.*

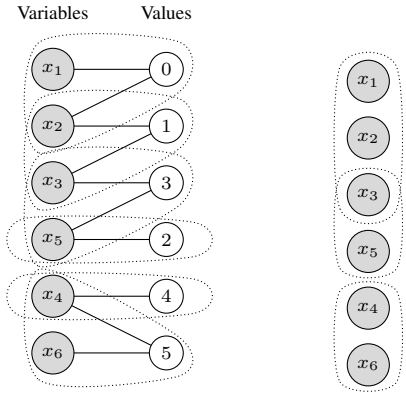
Proof: In any fully instantiated solution, two vertices of the intersection graph are neighbors if and only if the associated variables are instantiated to the same value. □

In general, f^{val}-decomposition does not preserve GAC neither (Theorem 5). It is worth noticing that any subset associated with a value is included in at least one subset associated with a vertex neighborhood in the intersection graph. Therefore, $f^{\cap}_{vn}$ can be seen as a compromise between $f^{\cap}_{cc}$ and f^{val}.

Theorem 5 *f^{val}-decomposable constraints are not all f^{val}-GAC-preserving.*

Proof: The f^{val}-decomposition of GCC amounts to its decomposition into a conjunction of OCCURRENCE constraints. This decomposition does not achieve GAC for GCC [11]. □

Overall, the f^{val}-decomposition is quite naive. Therefore, we extend it by introducing a new multifunction family $f^{val}_{\mathcal{P}}$, to enumerate some combinations of subsets associated with values. More precisely, any variable subset $\mathcal{Y} \in f^{val}_{\mathcal{P}}(\mathcal{X})$ is induced by $\mathcal{P}(f^{val}(\mathcal{X}))$. Such decomposition offers new tradeoff possibilities between filtering and runtime. An illustration over our running example is provided in Figure 4. In this example, we arbitrarily decided to group values into three pairs $(0, 1)$, $(2, 3)$ and $(4, 5)$, leading to variable subsets $\{x_1, x_2, x_3\}, \{x_3, x_5\}$, and $\{x_4, x_6\}$.



(a) $f^{val} = \{\{x_1, x_2\}, \{x_2, x_3\}, \{x_5\}, \{x_3, x_5\}, \{x_4\}, \{x_4, x_6\}\}$.

(b) A possible multifunction $f^{val}_{\mathcal{P}}(\mathcal{X}) = \{\{x_1, x_2, x_3\}, \{x_3, x_5\}, \{x_4, x_6\}\}$.

Figure 4: f^{val} and $f^{val}_{\mathcal{P}}$ illustration.

For instance, decomposing BINPACKING with an $f^{val}_{\mathcal{P}}$ multifunction enables to apply strong algorithms to subsets associated with some particular subsets of bins.

3 AN $f^{\cap}_{vn}$-DECOMPOSITION FRAMEWORK

We now suggest to put self-decomposition into practice, by introducing a general propagator which filters the $f^{\cap}_{vn}$-decomposition of any $f^{\cap}_{vn}$-decomposable constraint, in order to make propagation faster.

3.1 A simple graph-based propagation framework

Algorithm 1 Simple graph-based framework for $f^{\cap}_{vn}$-decomposition

```
// input
global Variable[] vars
global FilteringAlgorithm filter
// internal data structures
global int n
global Graph graph
global Set toCompute

    //called once by the solver, at root node
 1: method INITIALIZATION( )
 2:     // Initialisation of the intersection graph
 3:     n ← |vars|
 4:     graph ← ([1, n], ∅)
 5:     for (int i ∈ [1, n]) do
 6:         for (int j ∈ [i, n]) do
 7:             if (dom(vars[i]) ∩ dom(vars[j]) ≠ ∅) then
 8:                 graph ← graph ∪ {(i, j)} // adds edge (i, j)
 9:             end if
10:         end for
11:     end for
12:     GLOBALFILTERING( )
13: end method

    // called by the solver after one or many variable modification
14: method PROPAGATION( )
15:     while (toCompute ≠ ∅) do
16:         int i ← toCompute.POP( )
17:         // maintenance of the intersection graph
18:         for (int j|(i, j) ∈ graph) do
19:             if (dom(vars[i]) ∩ dom(vars[j]) = ∅) then
20:                 graph ← graph \ {(i, j)} // removes edge (i, j)
21:             end if
22:         end for
23:         // Filters locally, over vertex neighborhoods only
24:         filter.APPLYON({vars[j]|(i, j) ∈ graph})
25:     end while
26: end method

    // called by the solver anytime a variable domain is modified
27: method ONVARIABLEMODIFICATION(int i)
28:     toCompute ← toCompute ∪ {i}
29: end method

30: method GLOBALFILTERING( )
31:     filter.APPLYON(vars)
32:     toCompute ← ∅
33: end method
```

Algorithm 1 provides implementation guidelines for the $f^{\cap}_{vn}$-decomposition of any $f^{\cap}_{vn}$-decomposable constraint. The algorithm has two main steps: First, an initialization method (INITIALIZATION, lines 1 to 13) builds the intersection graph and filters all variable domains once (GLOBALFILTERING, lines 30 to 33) to propagate initial domains. Second, incremental propagations call the filtering algorithm of the constraint ($filter$) over vertex neighborhood subsets (PROPAGATION, lines 14 to 26). Such propagation is called by the solver after one or many variables had their domain filtered. The set of changed variables is updated through an *Observer* design pattern (ONVARIABLEMODIFICATION, lines 27 to 29), so that next propagations can filter on their neighborhoods. For any modified variable $x \in \mathcal{X}$, $filter$ is applied on the (maximal) subset of variables whose domain share at least one value with the domain of x.

As introduced in Section 2, this framework aims at obtaining a speedup on the propagation runtime when the fix-point is likely to be achieved without considering all variables. For instance, in case of a cubic algorithm, it may be preferable to perform many iterations on subsets of a few dozens or hundreds of variables than a single one over a thousand of variables.

Once we have the main structure of the framework, it is easy to design some hacks and tricks to enhance performances on either dedicated constraints or the general case. The next Section lists a few heuristics to improve the behavior of Algorithm 1.

3.2 Advanced implementation

This section provides several refinements of Algorithm 1 to improve its expected runtime.

Algorithm 2 Limitation of pathological cases

```
global final double α ← 2 // safety parameter (arbitrary value in [0, n])
 1: method PROPAGATION( )
 2:     int total ← 0
 3:     for (int i ∈ toCompute) do
 4:         total ← total + |graph.neighbors[i]|
 5:     end for
 6:     // Heuristic criterion to limit pathological cases
 7:     if (total > α.n) then
 8:         // The neighborhood-based decomposition may be a waste of time
 9:         GLOBALFILTERING( )
10:     else
11:         // same instructions as PROPAGATION method in Algorithm 1
12:     end if
13: end method
```

Algorithm 3 Conditional propagation

```
1: method ONVARIABLEMODIFICATION(int i)
2:     // Heuristic criterion to avoid some presumably useless propagations
3:     // The boolean function CONDITION(i) can be implemented as wished
4:     // (e.g., |dom(vars[i])| < 10, or even a random test)
5:     if (vars[i].INSTANTIATED( ) ∨ CONDITION(i)) then
6:         toCompute ← toCompute ∪ {i}
7:     end if
8: end method
```

First, the graph initialization can be improved by precomputing intersections of domain bounds with a sweep-line algorithm [5].

Second, a simple observation is that, in case too many variables had their domain changed, then it may be faster to filter over all variables once and disable the local filtering. The pathological example is to consider that every variable domain has been modified and that the intersection graph is complete, *i.e.,* any pair of vertices are neighbors. Within this configuration, the local filtering will trigger n times the filtering algorithm over all variables, which is clearly a bad thing. Thus, we suggest to reduce this risk, by adding a condition before triggering the local filtering, see Algorithm 2.

Last but not least, in many cases a high level of consistency is not worth when solving large, but not necessarily hard (from a combinatorial point of view) instances [10]. Within such consideration, we suggest to decrease the number of filtering calls by adding a condition to filter over the neighborhood of a variable which has been modified, see Algorithm 3. Such a condition can be related to the variable domain to detect cases where only a poor domain reduction can be expected from filtering. In the case of a set variable, it can be for instance the existence of a non-empty lower bound. Also, in case of a task variable (see Section 4), the condition can be the existence of a non-empty compulsory part, which amounts to the previous case. More generally, it can be any boolean test, even random. However, if the variable is instantiated, then the propagation must be performed to guarantee constraint satisfaction. Therefore, Algorithm 3 reduces calls to $filter$, while preserving a solution checker.

4 USE CASE: THE CUMULATIVE CONSTRAINT

This section illustrates the concept of global constraint self-decomposition with the CUMULATIVE constraint. We detail how to make this constraint self-decomposable. Next, we provide an experimental study which highlights the interest of our framework.

4.1 Making CUMULATIVE $f^{\cap}_{vn}$-decomposable

Because several variants exist, it is necessary to define precisely what we mean by CUMULATIVE. The cumulative constraint we consider states that the resource consumption of a given set of non-preemptive tasks should not exceed a given fixed capacity, at any point in time. A task includes a start time, a duration, an end time and a non-negative resource usage. If tasks are encoded with integer variables, which is the usual case, then CUMULATIVE is not $f^{\cap}_{vn}$-decomposable. However, with a higher level of abstraction over tasks, then $f^{\cap}_{vn}$-decomposition becomes possible. We consider a set of abstract variables $\mathcal{X}$ to represent tasks. We call them task variables. Thus, we use the following constraint signature: CUMULATIVE(`TaskVariable[]` $\mathcal{X}$`, Int Capacity`). We define task variable domain intersection as follows: Two task variables have a non-empty domain intersection if and only if, given their current domains, they may overlap in time and their resource usage may both be different from zero. Therefore, the intersection graph of CUMULATIVE is mainly related to time. As the resource must not be exceeded for every point in time, and as consuming tasks which overlap in time are neighbors with our definition of task domain intersection, then CUMULATIVE is $f^{\cap}_{vn}$-decomposable. This can be generalized to multiple resources. Furthermore, with a similar methodology based on abstract variables, then DISJUNCTIVE, DIFFN and GEOST are self-decomposable as well.

4.2 Practical interest

Scaling over cumulative and packing problems has been raised as a major issue for CP [1]. Self-decomposition is part of the solution. Indeed, it does make sense to have a lot of variables, but a sparse intersection graph. This happens when planning over a long term horizon: Time windows, precedences, and other side constraints, bring structure to the variable domains. Even if tasks remain to be precisely fixed in time, we may know the day (or week, month, *etc.*) it will occur. Therefore, it can be expected that only a small portion of variables have overlapping domains. To illustrate our point, we now evaluate the impact of self-decomposing CUMULATIVE. We compare its original implementation, which is propagated over all variables (`FULL`) with its $f^{\cap}_{vn}$-decomposition (`SELF`), implemented as described in Section 3.2. Our implementation has been integrated into `Choco-3.2`[2] so that the community can use it and reproduce our results. All the experiments were done on a `Mac Pro` with a 6-core `Intel Xeon` at 2.93 Ghz running on `MacOS 10.6.8`, and `Java 1.7`. Each run had one core and a five-minute time limit.

As a first illustration, we consider the minimum capacity cumulative problem. This problem consists of finding a schedule which satisfies CUMULATIVE and which minimizes the resource capacity. The initial capacity is unbounded, so it is trivial to find a first solution. However, finding an optimal solution is NP-hard. To highlight the impact of the proposed framework on scalability, we randomly generated 250 large CUMULATIVE instances. Each instance has n tasks, with $n \in \{1000, 5000, 10000, 15000, 20000\}$. For every task, we randomly generate a time window for its start, a fixed duration and a fixed height among respective intervals $[1, n]$, $[1, 50]$ and $[1, 5]$. Thus, this generator enables to get time-structured instances. As a CUMULATIVE filter, we use the state-of-the-art sweep algorithm[3] [7]. Even if finding a first solution can be trivially achieved with a greedy algorithm, a CP model may encounter scalability issue because the constraint is propagated at each search node. Therefore, we study the ability to find a feasible solution and to compute a good solution. Results are reported in Table 1. Regarding feasibility, `SELF` is able to compute a solution on all instances, whereas `FULL` does not solve any instance having between $15,000$ and $20,000$ tasks. Furthermore, `SELF` is more than ten times faster then `FULL`. It solves $20,000$-task instances in 57 seconds on average, whereas it takes 180 seconds for `FULL` to solve $10,000$-task instances. Within the same time, the self-decomposition enables to explore a much larger part of the search space. `SELF` explores up to ten times more search nodes than `FULL`. This enables the CP model to find better solutions. `SELF` improves the objective value by respectively 28.4% and 26.1% on average on $5,000$-task and $10,000$-task instances. Overall, the self-decomposition of CUMULATIVE brings substantial scalability improvement on this benchmark.

Table 1: Scalability comparison of a CUMULATIVE propagated over all variables (`FULL`) with its self-decomposition (`SELF`).

		Number of tasks				
		$1,000$	$5,000$	$10,000$	$15,000$	$20,000$
Nb. solved instances*	`FULL`	50	50	50	0	0
	`SELF`	50	50	50	**50**	**50**
Avg. time (s) first solution	`FULL`	1.4	36.2	179.9	-	-
	`SELF`	**0.5**	**3.0**	**10.8**	**27.9**	**57.1**
	$\frac{\texttt{FULL}}{\texttt{SELF}}$	**2.8**	**12.3**	**16.6**	-	-
Avg. search node count	`FULL`	164k	42k	15k	9k	6k
	`SELF`	**638k**	**392k**	**301k**	**165k**	**100k**
	$\frac{\texttt{SELF}}{\texttt{FULL}}$	**3.9**	**9.3**	**19.9**	**18.6**	**16.6**
Avg. objective value	`FULL`	85	124	142	-	-
	`SELF`	85	**89**	**105**	**129**	**137**
	$\frac{\texttt{FULL}-\texttt{SELF}}{\texttt{FULL}}$	**0.1%**	**28.4%**	**26.1%**	-	-

*: number of instances for which a feasible but not necessarily optimal solution could be computed within the five-minute time limit.

Next, we investigate the interest of self-decomposition on an industrial scheduling problem[4]. The problem is to find a feasible schedule for a set of tasks which satisfies a set of precedence constraints and which never exceeds the capacity of any resource. There are 8 resources and between $6,000$ and $16,000$ tasks. Unlike the previous example, task time windows are initially very large. Results are reported in Figure 5. As can be seen, `SELF` is always much faster than `FULL`. On average, the $f^{\cap}_{vn}$-decomposition of CUMULATIVE brings a speedup of 5 on this benchmark.

[2] http://www.emn.fr/z-info/choco-solver/

[3] The greedy mode is not used as it removes feasible values and solutions.

[4] Private communication with Arnaud Letort and Helmut Simonis

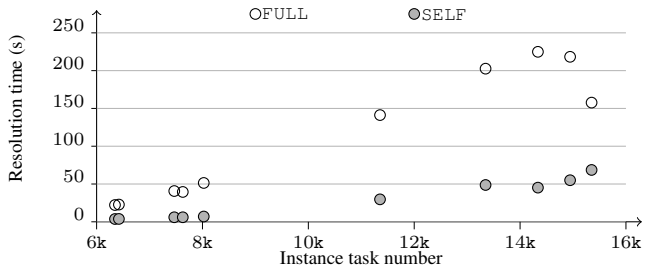


Figure 5: Impact of self-decomposition on an industrial scheduling problem. Each point represents the resolution of an instance.

Even on small problems, self-decomposition presents a practical interest : it may capture local trends, such as peak or low activity periods, which are common in scheduling. This occurs in the jobshop problem of the MiniZinc distribution[5]. Such problem involves no restrictive time windows but precedence constraints and several resources. The schedule make span must be minimized. To solve this problem, we use the CUMULATIVE filtering algorithm that is provided by default in `Choco-3.2`. It includes some partial energy-based reasoning which strengthens the filtering without introducing a significant overhead. As can be seen in Figure 2, the decomposed approach is more efficient than the classical one. This is not due to a faster propagation but a stronger filtering : whenever applied to peak periods, the partial energy-based filtering is able to detect unfeasibilities early, whereas its application to the whole set of variables brings no inference, because peaks are compensated by low activity periods.

Table 2: Resolution of a jobshop problem. Comparison of the best solution found by `FULL` and `SELF`, within a five-minute time limit.

Instance	Best solution value	
	`FULL`	`SELF`
jobshop_abz6.fzn	985	985
jobshop_la01.fzn	816	816
jobshop_la02.fzn	764	**733**
jobshop_la03.fzn	758	**714**
jobshop_la04.fzn	682	682
jobshop_la05.fzn	593*	593*
jobshop_mt06.fzn	55	**55***
jobshop_mt10.fzn	1092	**1084**

* : Optimality has been proved

5 CONCLUSION

We have introduced the definition of self-decomposable constraints, a large family of constraints which can be propagated over variable subsets only. It includes at least ALLDIFFERENT, SOMEDIFFERENT, DISJUNCTIVE, DIFFN, CUMULATIVE, GEOST and, under some assumptions, GCC and BINPACKING. We have formalized two simple well known decomposition patterns, $f^{\cap}_{cc}$ and f^{val}, which we have extended in order to obtain interesting tradeoff possibilities between filtering and runtime, leading respectively to $f^{\cap}_{vn}$ and $f^{val}_{\mathcal{P}}$.

In addition, we have proposed a generic framework to perform the $f^{\cap}_{vn}$-decomposition of a constraint. It takes advantage of the domain structure to make filtering algorithms incremental, in order to reduce the propagation runtime. This framework is equipped with a trigger which enables to detect and avoid some cases where the decomposition is presumably not worthwhile. More generally, as self-decomposition is dynamic, it can be turned on and off anytime.

[5] https://github.com/MiniZinc/minizinc-benchmarks/blob/master/jobshop2

Furthermore, we illustrated this concept on the CUMULATIVE constraint, which may involve numerous variables over large dimensions, but where only a reasonable amount of variables may be close to each others. Preliminary results show the interest of such approach for solving large scale problems and to capture local phenomena. Future work may extend this experimental study to packing constraints. Moreover, it would be interesting to investigate the benefit of our framework within a Large Neighborhood Search (LNS) [13], which is a common approach to solve large optimization problems. Presumably, self-decomposition should enable to skip most fixed variables, hence bring a significant improvement in runtime.

However, our approach is somehow against the tide, because most recent works on scalability suggest *meta*-global constraints (*e.g.,* [7]), which represent conjunctions of global constraints, whereas we suggest a dynamical decomposition. As we keep existing filtering algorithms, we believe our approach is easier to implement.

Finally, while this work stems from filtering operational considerations, it also has a theoretical dimension. We believe the concept of self-decomposition may be reused for different purposes, such as guiding search or conflict analysis.

ACKNOWLEDGEMENTS

The authors thank the referees for their remarks as well as Arnaud Letort and Helmut Simonis for providing us an industrial data set.

REFERENCES

[1] Panel of the Future of CP, 2011. Perugia, CP'11.
[2] S. Asaf, H. Eran, R. Richter, D. P. Connors, D. L. Gresh, J. Ortega, and M. J. Mcinnis, 'Applying constraint programming to identification and assignment of service professionals', in *CP*, volume 6308 of *LNCS*, pp. 24–37. Springer, (2010).
[3] R. Barták, 'Dynamic global constraints in backtracking based environments', *Annals of Operations Research*, **118**(1-4), 101–119, (2003).
[4] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit, 'Global Constraint Catalog: Past, Present and Future', *Constraints*, **12**(1), 21–62, (2007).
[5] J. L. Bentley and T. A. Ottmann, 'Algorithms for reporting and counting geometric intersections', *IEEE Trans. Comput.*, **28**(9), 643–647, (1979).
[6] C. Bessiere, G. Katsirelos, N. Narodytska, and T. Walsh, 'Circuit Complexity and Decompositions of Global Constraints', in *IJCAI*, Constraints, Satisfiability, and Search, pp. 412–418, (2009).
[7] A. Letort, M. Carlsson, and N. Beldiceanu, 'A synchronized sweep algorithm for the *k-dimensional cumulative* constraint', in *CPAIOR*, volume 7874 of *LNCS*, pp. 144–159. Springer, (2013).
[8] M. J. Maher, 'Open contractible global constraints', in *IJCAI*, pp. 578–583. Morgan Kaufmann Publishers Inc., (2009).
[9] U. Montanari, 'Networks of constraints: Fundamental properties and applications to picture processing', *Information Sciences*, **7**(0), 95 – 132, (1974).
[10] L. Perron, 'Operations research and constraint programming at google', in *CP*, volume 6876 of *LNCS*, p. 2. Springer, (2011).
[11] J.-C. Régin, 'Generalized arc consistency for global cardinality constraint', in *AAAI/IAAI*, volume 1, pp. 209–215. AAAI Press / The MIT Press, (1996).
[12] C. Schulte and G. Tack, 'Weakly Monotonic Propagators', in *CP*, volume 5732 of *LNCS*, pp. 723–730. Springer, (2009).
[13] P. Shaw, 'Using constraint programming and local search methods to solve vehicle routing problems', in *CP*, volume 1520 of *LNCS*, pp. 417–431. Springer, (1998).
[14] P. Van Hentenryck, Y. Deville, and C.-M. Teng, 'A generic arc-consistency algorithm and its specializations', *Artificial Intelligence*, **57**(2-3), 291–321, (1992).
[15] T. Walsh. Public remark, 2012. Quebec city, CP'12.